1988-09

Rapidly Prototyping Real-Time Systems

Luqi

IEEE

Rapidly Prototyping Real-Time Systems

Luqi and *Valdis Berzins*, *Naval Postgraduate School*

Prototyping approaches that use modularity and reusable components are both promising and practical. The method described here is such an approach for large real-time developments.

The demand for large, high-quality systems has increased to the point where a jump in software technology is needed. Rapid prototyping is one of the most promising solutions to this problem. Rapid prototyping is particularly effective for ensuring that the requirements accurately reflect the user's real needs, increasing reliability and reducing costly requirements changes.

This article presents a method for rapidly constructing executable prototypes for large real-time systems. In our method:

- The prototype must satisfy and be traceable to its requirements. We construct prototypes iteratively to analyze and strengthen the requirements.
- The prototype must be easy to modify because it will be subject to many revisions before the user is satisfied with the requirements as reflected by the prototype's behavior.
- The prototype code must be easy to read and analyze because the prototype is supposed to support analysis of the intended system and document an initial design. Clear and simple high-level structures let designers answer questions easily about the system's properties and feasibility. We developed our prototyping method with the Prototype-System Description Language[1] and an automated prototyping environment.[2]

The goal of a prototype is different than that of a production software system. Efficient use of designer time and rapid feedback are more important than robust operation, efficient use of machine resources, and completeness.

Our approach

We use an integrated approach to prototyping that combines a computational model tailored for real-time systems with a high-level prototyping language, a systematic design method for rapid prototype construction, and an automated prototyp-

0740-7459/88/0900/0025/$01.00 © 1988 IEEE

Related work

There are two main types of system decomposition: one based on dataflow and the other on control flow.[1] We know of no previous work providing a syntactic and semantic way to combine dataflow and control flow for system design.

Several nonprocedural languages have been proposed in recent years. These languages are easy to analyze and expose an algorithm's natural parallelism. The design of the nonprocedural control constraints in PSDL owes much to these ideas. But one difference between our work and previous approaches to rapid prototyping using applicative languages is that we provide a black-box specification for each component in addition to an implementation. Black-box specifications state which properties of the prototype are required in the intended system and can also be used to retrieve reusable components from a software base.

Our approach to execution support for PSDL owes much to the pioneering work on real-time system modeling and scheduling by Liu and Mok.[2] Their work shows how asynchronous tasks can be modeled by synchronous equivalents.

There has been a fair amount of work on machine-aided rapid prototyping for systems without hard real-time constraints. For example, Wasserman and colleagues have described a system to make prototypes of user interfaces for interactive systems.[3]

Bruno and Marchetto have used petri nets to prototype the synchronization and interprocess-communication aspects of process-control systems.[4] While the notation is not very easy to read, it does support automated deadlock detection and performance evaluation in terms of steady-state probabilities for graph markings.

Cameron has described a technique for modeling real-world systems that is appropriate for typical data-processing applications,[5] but his method does not address real-time constraints and is weak on data abstractions.

Many informal versions of dataflow diagrams have been used extensively to model the data-transformation aspects of software systems. Dataflow diagrams are easy to read and reveal the internal structure of a process and the potential parallelism inherent in a design, making dataflow attractive to designers.

We believe an automated prototyping environment should be able to graphically display and update the prototype's system structure. However, these informal notations do not provide a unified mechanism to represent all the relevant attributes of software systems (such as timing and control) and are not sufficiently formal to be executable.

A more precise model of a dataflow computation has been developed for hardware design, and we have extended the model and the notation to include control aspects and critical timing constraints in a two-dimensional dataflow diagram without losing its natural benefits.[6] These extensions are needed for the design of systems with hard real-time constraints.

References

1. K. Iwamoto and O. Shigo, "Unifying Dataflow- and Control-Flow-Based Modularization Techniques," *Proc. Fall Compcon*, CS Press, Los Alamitos, Calif., 1981, pp. 271-277.
2. A.K. Mok, "The Decomposition of Real-Time System Requirements into Process Models," *Proc. Real-Time Systems Symp.*, CS Press, Los Alamitos, Calif., 1984, pp. 125-133.
3. A. Wasserman et al., "Developing Interactive Information Systems with the User Software-Engineering Methodology," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 326-345.
4. G. Bruno and G. Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process-Control Systems," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 346-357.
5. J. Cameron, "An Overview of JSD," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 222-240.
6. Luqi, *Rapid Prototyping for Large Software-System Design*, PhD dissertation, Computer Science Dept., Univ. of Minnesota, Minneapolis, Minn., 1986.

ing environment that lets designers effectively use a base of reusable software components.

The computational model prevents hidden interactions among system components and encourages designs with good module independence. The language supports the model and combines it with a powerful set of data and control abstractions to make it easy to describe systems at a high level. The automated environment relies on a software-base-management system for retrieving and adapting reusable components.[3]

Strategy. We use a problem-oriented, top-down strategy to focus the prototyping effort on a critical problem or on selected attributes of the entire system. The major system attributes that a prototype must demonstrate to the user usually appear in a critical subsystem, so you must create a skeleton of the intended system with which the fully prototyped subsystem must interact. This skeleton helps you at least partially simulate the critical subsystem's environment to demonstrate the prototype's behavior. You also need an initial description of the intended system as a basis for discussion. An interactive graphics editor lets you build this skeleton rapidly and understand it quickly.

During the prototyping effort's iterations, our environment lets you update the prototype quickly and easily.

The prototyping method associated with PSDL results in a hierarchically structured prototype. The method gives you a decomposition strategy for filling in more details at any level of the prototype design. It helps you focus on the critical subsystems that must be refined to resolve the problems that required the rapid prototyping effort in the first place.

The prototype design is based on abstract functions, abstract data, and abstract control. This high-level view emphasizes the overall configuration at each level without getting embroiled in low-level details. PSDL supports functional, data, and control abstractions, which hide lower level details in the prototyping method. Only the concepts directly used at each level appear in the abstractions of the dataflow diagram at that level.

Language. We recognize that a good language for expressing design in a precise model is essential for rapid prototyping. PSDL was designed to serve as an executable prototyping language at the specification or design level.

Our prototyping method produces a PSDL-format description of the prototype. PSDL provides sufficient structures and descriptive ability to express the internal and external features of the system modules. PSDL also uses a clear and powerful modularization model to build and describe the prototype. The model is based on dataflow under real-time constraints and uses an enhanced dataflow diagram that includes nonprocedural control constraints and timing constraints.

PSDL and its prototyping method were designed primarily for *hard* real-time systems. (A hard real-time constraint is a bound on the response time of a process or the period between invocations that must be satisfied under all operating conditions. A hard real-time system has hard real-time constraints as part of its requirements.) The box above left describes related work that PSDL benefited from.

Decomposition

Problem decomposition is central in the design of any large system. We propose a uniform decomposition method that combines the advantages of dataflow and control-flow decomposition and helps the designer achieve good modularity.

Good modularity is a key factor for increasing productivity because it reduces the debugging effort to produce a correct executable system and because it improves the understandability, reliability, and maintainability of the developed system. These features are especially important in rapid prototyping.

Combining dataflow and control flow. Each criterion for decomposing a system is based on a computational model; dataflow and control flow are two popular decomposition criteria. The components of a dataflow decomposition are independent sequential processes that communicate through buffered data streams, while the components of a control-flow decomposition are procedures that are called by and return to a main procedure with a single control thread.

Which decomposition strategy to choose depends on how much control over the sequencing of suboperations is necessary or desirable. Iwamoto and Shigo[4] have suggested circumstances where each decomposition is preferred and have given some restrictions sufficient to guarantee that the computed results are independent of scheduling decisions. Their system does not address real-time constraints and is a relatively low-level extension to Fortran applications that is subject to many confusing restrictions.

They use dataflow decomposition where there is a mismatch between the structures of an operator's input and output data stream, introducing an intermediate data stream of lower level data elements to resolve the structure clash. An example is a decomposition with a dispatch operator that recognizes several alternative kinds of inputs and routes them to the appropriate special-purpose operator. Dataflow decomposition for such a structure requires that the data elements contain extra sequencing information to make sure the result streams do not get out of order when they are merged, because you cannot predict the relative speeds of independent processes under the usual interpretation of dataflow.

Iwamoto and Shigo use a control-flow decomposition where the data stream forks into several branches and is rejoined and where the operators on the branches influence each other's results through state changes (in these cases, dataflow decomposition will result in computations whose results can depend on the unpredictable behavior of the process scheduler). An example is a transaction with multiple updates to a shared database, where the final state of the database may depend on the arbitrary order of the updates performed by operators on parallel branches of the dataflow graph.

We propose a uniform decomposition method that combines the advantages of dataflow and control-flow decomposition and helps achieve good modularity.

Control-flow decompositions tend to overly restrict the sequencing of suboperations, while dataflow decompositions sometimes lack semantically necessary restrictions on that sequencing. To avoid these problems with dataflow decompositions, we have developed a new underlying model of computation for PSDL, which is based on dataflow and guarantees that the results of a computation do not depend on undetermined properties of the schedulers. Our method combines control constraints with the dataflow model to achieve the best modularity with sufficient control information. We use dataflow to simplify the interactions between modules, eliminating direct external references and communication through side effects.

The first problem in Iwamoto's and Shigo's approach — that you must provide extra sequencing information — does not arise in our model because a PSDL rule says that a composite operator cannot fire again until all the internal activity associated with the previous firing is complete. This rule provides a kind of mutual exclusion that prevents interference between successive actions by the same operator without preventing concurrent execution of a composite operator's components.

Their second problem with dataflow decomposition does not arise in PSDL prototypes because there is no implicitly shared changeable data.

PSDL computation model. The enhanced dataflow diagram that PSDL is based on is a directed graph with associated timing and control constraints. The nodes of the graph are operators; the arcs are data streams.

Operators are either functions (without an internal state) or state machines (with an internal state). When an operator fires, it reads one input value from each incoming arc and puts at most one computed output value on each outgoing arc. An operator's firing can be triggered by the arrival of a specified set of input data values or by a periodic timing constraint.

The firing of an operator and the production of an output value can also be subject to conditional control constraints that depend on locally available data values. This limited facility for interconnecting operators is well-matched to the needs of real-time systems, where each operator must complete its task in a fixed time.

A data stream carries values of an abstract data type. Both the built-in and user-definable PSDL data types are immutable. An immutable type has no operations for changing the state of a data object, so all changes appear as newly generated data values rather than as updates to existing data objects.

The generic built-in PSDL types include tuples (records), one-ofs (tagged variants), sets, sequences, maps (lookup tables), and relations. These types provide a powerful facility for defining finite collections of any value type and make it easy to construct many user-defined abstract data types. PSDL also has conventional data types for numbers, strings, and truth values.

Each data stream is either a dataflow stream, which guarantees that each data

element that enters is delivered exactly once, or a sampled stream, which guarantees that a data element can always be entered into or delivered from the stream on demand, at the cost of replicating elements or discarding older values. A dataflow stream acts like a first-in, first-out queue whose length is bounded by 1. A sampled stream acts like a memory cell that always contains the most recent data value in the stream and that can be updated at any time.

In PSDL, the control and timing constraints of the operator receiving a stream determine whether the stream is a dataflow or sampled stream. (Dataflow streams are discrete dataflows; sampled streams are continuous dataflows.) The constraints guarantee there will be data values on all the input streams of an operator whenever it fires. Exceptions are treated as data values of a special data type, which flow down data streams subject to the same rules as ordinary data values.

In PSDL, each operator can have a maximum execution time and a maximum response time, which are treated as hard real-time constraints. Operators with real-time constraints are periodic (synchronous) or sporadic (asynchronous). The firing frequency of each periodic operator is specified by giving its period. The minimum period between firings is also specified for each sporadic operator to record the necessary assumptions about worst-case operating conditions for asynchronous external events.

You can also associate control constraints with operators. These include conditions that act as guards for firing an operator or passing an output value to a data stream, and can control exception conditions or timers.

It is easy to describe individual timing constraints of a real-time system, but large real-time systems often contain a mixture of periodic and sporadic operators with many different frequencies. The interactions between such timing constraints can be very complex and very difficult to analyze without the help of a computer.

Hierarchical decomposition. The PSDL prototyping method develops a hierarchical design through stepwise refinement. While the model is created mostly in a top-down fashion, the process is guided by a tool that browses through the reusable components in the software base. Each operator and each data type associated with the data streams is given a black-box specification and subjected to further refinement. There are three possibilities for this refinement:

1. A search of the software base succeeds in retrieving a reusable component whose specifications match those of the required operator or type. The usefulness of partial matches is enhanced by facilities for instantiating generic reusable components and by PSDL control constraints, particularly conditional guards that can limit the execution of a reusable component to the cases where its behavior satisfies the requirements of the needed prototype component. In this case, all required lower level details are supplied from the software base and no further effort is required of the prototype designer.

The PSDL prototyping method develops a hierarchical design through stepwise refinement. It is created mostly top-down, guided by a tool that browses the reusable software base.

2. No match is found in the software base and the behavior of the needed component is sufficiently complex to decompose into a network of simpler types and operators. The component is refined as a lower level dataflow model, and the process is repeated recursively. As in any design process, the designer's skill and experience determine the speed and quality of the decomposition. PSDL contains a powerful set of built-in data types and control constraints to aid this decomposition.

3. No match is found in the software base and the behavior of the needed component is so simple that further decomposition would not be useful. This means the software base is incomplete. (This should not occur often for applications with a mature software base.) In this case, you would code a small, special-purpose module in the underlying language and add it to the design database containing the prototype. If you expect to develop similar systems, you can also add the code to the software base. You extend the software base with the new module *after* the rapid-prototyping effort is completed because you must generalize the module, develop a good specification for it, and certify its correctness before adding it — these are time-consuming prerequisites that you likely will want to do after you have completed your current prototype.

Locality and component scoping. PSDL has been designed to prevent implicit interactions between operators, thus encouraging model independence. Because there is no global data in PSDL, operators must rely on incoming data streams for all input. Because all PSDL data values are immutable, operators cannot interact through state changes in a shared mutable data object. Two PSDL operators cannot interact through state changes unless both have explicit dataflow connections to the same state machine.

The state of a state-machine operator is purely local in PSDL. The operator can be influenced only by sending data values to one of the operator's input streams through a local name-scoping rule for composite operators. Access to state machines must be local; you cannot send a data stream directly to a composite operator's component because the component names are not visible outside the implementation part of the composite operator. For the same reason, two composite operators cannot share the same instance of a state machine as a subcomponent (although they could both use different instances of the same generic state machine).

This locality makes it easier to modify PSDL prototypes because the number of modules affected by a change is limited and can be determined by a straightforward mechanical analysis of the prototype's dataflow structure. It also makes it easier to distribute the parts of the computation among several processors, since there are no implicit interactions (which are difficult to implement in a loosely

coupled architecture).

The localized nature of the PSDL computational model encourages the prototype designer to first specify abstract state machines or functions and then to decompose them into loosely coupled networks of independent operators, which is the preferred structure for distributed software.

Supporting environment

An automated support environment is essential for the rapid construction of prototypes. PSDL and its prototyping method have been designed for use in an environment containing a software-base-management system, a syntax-directed editor with graphics capabilities, a design database, and an execution-support system.

Software base. The software-base-management system stores and retrieves reusable components. In addition to implementation information, each component in the software base must have a PSDL specification. The PSDL specification is organized as a set of distinct but related attributes. The software-base manager must provide component retrieval based on partial matches of these attributes, as well as a browsing capability similar to the one provided by the Smalltalk environment and a set of operators for tailoring and instantiating generic components.[2,3]

We assume that a sufficiently large, practical software base containing high-quality reusable components is attainable. It is important to have a relatively complete set of general-purpose components to perform the functions common to many systems, such as managing displays, sorting and searching, parsing input strings, and managing look-up tables. Many of these functions can be effectively encapsulated in a small set of abstract data types. It is very important to provide generic versions of the reusable components because it would otherwise be impossible to design with abstract data types while relying on standard reusable components for performing common utility functions.

Designer interface. The designer interface consists of a syntax-directed editor for PSDL and a graphics tool to construct and display dataflow diagrams. The syntax-directed editor helps speed the design process by eliminating syntax errors, automatically supplying keywords, and prompting you with a choice of legal syntactic alternatives at each point.

The graphics tool is a part of the syntax-directed editor that provides a graphical view of the dataflow diagram part of a composite module's PSDL implementation. It helps you visualize the relationships between the components of a decomposition through a two-dimensional dataflow diagram and provides a convenient way to enter and update the decomposition information in the enhanced dataflow diagram, which is part of a PSDL implementation of a component. This capability is important because the textual form of a dataflow diagram is harder to understand than the graphical form.

An automated support environment is essential for the rapid construction of prototypes. PSDL is designed to be used in an environment that has a software base, a syntax-directed editor, and a design database.

Design database. The design database in the prototyping environment contains PSDL designs. Using a database rather than a text file simplifies the job of writing programs that analyze PSDL prototypes and helps provide a continuous cross-referencing capability.

This cross-referencing capability is most important for requirements tracing and is used mostly in updating the requirements and adjusting the prototype to match. In this case, the binary relationship is satisfies-requirement. The design database must support retrievals of the form

- given a requirement, find all the PSDL components that implement it, or
- given a PSDL component, find all the requirements it implements.

Execution-support system. To construct and update a prototype rapidly, the execution-support system for PSDL must be efficient. Because prototype modifications are at least as frequent as prototype runs, both preprocessing time and execution time must be given roughly equal weight, making interpretive implementation preferable to compilation.

The execution-support system should be able to save the state of a computation and to run several alternative versions of a prototype from a state without repeating the initial part of the computation. This is important because the designer will have a dialogue with the user during which an aspect of the prototype's behavior is demonstrated and criticized and during which alternatives are explored.

Because it may have taken a long user interaction to arrive at the state to be examined, it is not acceptable to require the designer and the user to go through many repetitions of that dialogue or even to rerun the initial part of the dialogue from a saved script. The need to modify the prototype in the middle of a run implies the need for a dynamic loader and for some way to rapidly respond to changed specifications.

Our execution-support system consists of a static scheduler, a dynamic scheduler, and a debugger.[5]

The static schedulers schedules time for the computations with hard real-time constraints so that all the timing constraints are guaranteed to be met. We use the standard approach of statically allocating time slots sufficient for the worst-case execution times of the operators. The abstract treatment of timing information is an important property of the dataflow model because only the essential time orderings among the events in the computation are given. These time orderings act as constraints on the static scheduler and allow the flexible exploration of schedules for multiprocessor configurations.

The dynamic scheduler schedules the computations that do not have hard real-time constraints in time slots not used by the time-critical computations.

The debugger exercises the prototype, collects statistics, and lets you readily modify it to conform to new or modified requirements.

Prototype construction

We have chosen a real-world example — a hyperthermia system — to demonstrate our design method. We chose this example because the software used for temperature control in the hyperthermia system is a typical embedded system with hard real-time constraints and because the application is significant and realistic. It is large enough to demonstrate the essential features of large-scale prototype programming but not too large to describe here.

The problem. One approach to combatting cancer is to destroy tumorous cells selectively with heat. One way to do this is with a hyperthermia system, which uses a microwave generator connected to a fine tuner and matching control system to produce and deliver controlled, local heating directly to tumors. A computerized control system adjusts power output automatically to maintain the temperature in accordance with the treatment plan.

The hyperthermia system has four subsystems: a computer system, an operator's panel, a microwave generator, and a temperature sensor. The critical subsystem is the software that receives input from the temperature sensor and produces control commands to operate the whole system. The software controls the rest of the system, which is typical of real-time embedded systems. To demonstrate the prototype's behavior, we had to simulate the properties of the rest of the system.

The software subsystem's informal requirements, which are rough and typical of the initial requirements supplied by a user, were:

- Accept input tumor data in the patient's medical record from an existing source.
- Prepare the probes and their corresponding structures in the microwave and temperature-sensing systems.
- After the preparation is completed, have the power generator begin generating microwaves and then have the software control system adjust the intensity of the microwaves sent based on inputs from probes in the temperature-sensing system. The adjustment should be made according to the data describing the microwave-temperature-time pattern.
- The desired hyperthermia temperature for the therapeutic treatment is 42.5°C. The system should reach the indicated temperature in less than five minutes to leave sufficient time for treatment.
- After the system reaches the indicated temperature, it should keep the temperature stable for 45 minutes to kill tumorous cells. During this time, the treatment system should adjust the intensity of microwaves to keep the temperature stable with an error tolerance less than 0.1°C.
- The software subsystem must appropriately control the other subsystems of the hyperthermia system to ensure their correct operation.

Initial steps. Before building a prototype, we first analyzed the problem to decide what questions the prototype is supposed to answer, to identify which parts and attributes of the system to prototype, and to get the requirements for the prototype. Figure 1 shows the prototyping life cycle and the steps for updating the requirements.

The first step in our prototyping method is to determine which questions are supposed to be answered using the prototype. Typical questions that can be answered using a prototype are whether the proposed system behavior meets user needs, whether system I/O interfaces are acceptable, and whether proposed real-time constraints can be satisfied.

In the hyperthermia example, the questions we addressed were whether a real-time control system satisfying the requirements was feasible and whether the proposed control system was safe for use in hospitals.

The next step is to determine which part of the system must be prototyped to answer these questions. In the hyperthermia system, the critical subsystem was the software, since the feasibility and safety of the temperature probes and microwave generator were not in doubt.

The critical subsystem has interfaces to the doctor, temperature probes, and microwave generator. The attributes that affect safety and thus must be included in the prototype were the treatment temperature and the treatment time. The relation between the microwave power level and the treatment temperature must be determined and simulated to allow the evaluation of the proposed control algorithms for the microwave power level.

The third step is to rewrite the prototyping system's requirements into a clear and brief form because the initial English description is usually long, redundant, and imprecise. PSDL assumes that the requirements are structured as a set of named items. The PSDL facility for recording the correspondence between the requirements and the parts of the prototype works best if each item in the requirements represents a single constraint and if different items represent independent constraints. You can use more formal notation in PSDL, but we used the following:

- Shutdown: Microwave power must drop to zero within 300 ms of turning off the treatment system's switch.
- Temperature tolerance: After the system stabilizes, the temperature must be kept between 42.4°C and 42.6°C.
- Maximum temperature: The temperature must never exceed 42.6°C.
- Start-up time: The system must stabilize within five minutes of turning on the treatment system's switch.
- Treatment time: The system must shut down automatically when the tempera-

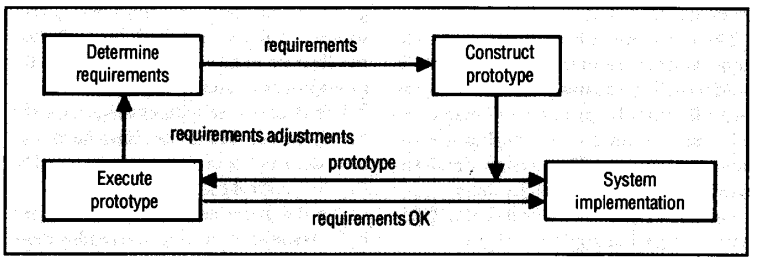


Figure 1. Prototyping life cycle and the steps for updating the system requirements.

ture has been above 42.4°C for 45 minutes.

Level of detail. One of the most important concerns is how much detail should be included in the prototype. Our guideline is to include the minimum amount of detail implied by the previous choice of what questions to answer and which part of the system to prototype. At this point, the critical things to keep in mind are:

- A prototype system is not a production system. The purpose of a prototype is to provide answers to questions about the requirements and the properties of the proposed system. The prototype must include only the aspects of system behavior relevant to answering these questions. It does not have to be a complete, reliable, or efficient realization of the proposed system.
- For the attributes and subsystem to be prototyped, you do not have to design the prototype in as much detail as you would the intended system. You can use functional simulation to reduce the amount of detail that must appear at a specific working level. You can leave out aspects that are not related to the questions or represent them with low-cost mock-ups.

For example, if the purpose of the prototype is to determine the effectiveness of a proposed control algorithm, the display formats are not critical and off-the-shelf defaults can be used. Conversely, if the central questions are related to human factors, the format and placement of the displays may have to be represented in detail, while the data content is not critical and can be filled in by a random-data generator. Extraneous details can be treated as lower level attributes and left to be realized in lower level components if the decomposition is eventually refined further.

There are three important considerations in determining the level of detail:

- Choose a minimum set of subcomponents at each level of the decomposition hierarchy.
- Eliminate unnecessary decomposition if reusable components of the same or nearly the same specifications can be found.
- Try functional simulation in the underlying programming language for the components whenever it can simplify the specification implementation.

Prototype components. PSDL prototype components are either operators or abstract data types. Every component will eventually have both a specification and an implementation part in PSDL. Specifications are developed for all components at a given level of the hierarchy before any of the implementation parts are considered, and the process is repeated until no more decomposition is needed.

The function of each component is clarified by writing its formal description and attribute specifications. Informal English descriptions are written for each component as design documentation if the formal descriptions and the attributes specifications are insufficient to describe the components or the design. The specification part of the top-level operator in the hyperthermia example was:

A prototype system is not a production system. It does not have to be complete, reliable, or efficient.

```
OPERATOR
  Brain_Tumor_Treatment_System
SPECIFICATION
  INPUT patient_chart: medical_history,
    treatment_switch: boolean
  OUTPUT treatment_finished: boolean
  STATES temperature: real
    INITIALLY 37.0
  DESCRIPTION
  { The brain-tumor treatment system kills
    tumorous cells using hyperthermia
    induced by microwaves. }
END
```

This operator is a state machine. The only component of the state needed for the prototype is the temperature of the tumor, which is specified to be normal body temperature in the initial state.

Medical_history is an abstract data type appearing as an external input to the system. A partial PSDL specification for this data type is given below. The complete data type has many other operations, but only those related to the brain-tumor treatment system are included in the prototype. This illustrates the principle of including only those details needed for the purposes of the prototype.

```
TYPE medical_history
SPECIFICATION
  OPERATOR Get_Tumor_Diameter
  SPECIFICATION
    INPUTS patient_chart:
      medical_history,
      tumor_location: string
    OUTPUTS diameter: real
    EXCEPTIONS no_tumor
    MAXIMUM EXECUTION TIME 5 ms
    DESCRIPTION
    { Returns the diameter of the tumor at
      a given location, produces an excep-
      tion if no tumor at that location. }
  END
  KEYWORDS patient_chart,
    medical_record,
    treatment_record, lab_record
  DESCRIPTION
  { The medical history contains all the
    disease and treatment information for
    one patient. }
END
```

Decomposing components. After the components have been identified and specified, you search the software base to determine if they match existing reusable components. If the retrieval results in a reusable component with sufficiently close specifications, the implementation is finished. Otherwise, you decompose complex components into more primitive parts and code simple ones in the underlying language.

You decompose an abstract data type by giving a representation for the values of the type in terms of other simpler types and then decomposing the operators of the type. You decompose an operator by expressing it as a dataflow decomposition involving simpler types and operators. You do this by identifying some useful lower level operators, guided by the concepts found in the problem description and requirements, and by considering the operators available in the software base.

You can find such operators through the browsing tool and by retrieving inexact matches to a PSDL specification from the software base. You next determine the interconnections of the operators and the types of the data streams. You add control constraints where needed, and you then allocate the timing constraints of the composite component its parts at the next lower level.

The operator for Brain_Tumor_Treat-

ment_System was not available in the software base. We chose to decompose the operator because it was not a simple one.

Verifying requirements. To check that the requirements for maintaining the treatment temperature were met, it was useful to divide the prototype into a control system and a simulation of the system to be controlled. In this case, the system to be controlled consisted of the patient, the microwave generator, and temperature-measurement system. The only attributes of this system needed for the prototype were the temperature reading and the desired power level for the microwave generator.

Figure 2 shows the data streams in the PSDL implementation and the lower level operators.

The Brain_Tumor_Treatment_System is described as a periodically executing feedback loop that implements a state machine. Each cycle in a PSDL dataflow diagram must contain a state variable with a declared initial value. In the example, the only cycle contains the state-variable temperature.

We chose the period of the Brain_Tumor_Treatment_System to meet the emergency-shutdown requirement, which requires the system to set the power to zero within 300 ms after the treatment system's switch is turned off. This requirement will be met if the sum of the period and the maximum execution time of the hyperthermia system do not exceed 300 ms.

Because the treatment system's switch can change unpredictably, it can be almost a full period before the system samples the switch's value. Hyperthermia_System must then be executed before a response to the changed input signal can be generated. A tighter time bound cannot be established without looking inside the hyperthermia system, which we want to avoid to preserve the prototype's hierarchical structure.

The control function is very important for the patient's safety, and the temperature tolerances are tight compared with the accuracy of common thermometers, so ample time must be allocated for the accurate computation of treatment_power. Because it takes time for the tumor to heat up in response to a higher power level, the system must allow enough delay for Simulated_Patient to respond to each power-level change to avoid control instabilities in the actual system. Because both functions have roughly equivalent demands for time, we allocate equal time periods for both components.

Because this was the top level of the design, there were no other processes competing for computing resources, so we could use the entire period to execute the two functions. These considerations led to a period of 200 ms, with maximum execution times of 100 ms for each component. We could have improved the accuracy of these timing values by measuring the running time of the prototype or by using a static timing-analysis tool to calculate worst-case time bounds for loop-free code (using the instruction times of a particular compiler and machine) and constructing an accurate simulation of the power-temperature-time relationship for the human brain.

The period must also let the system adjust the power level fast enough to guarantee that the temperature remains in the allowable range. The correspondence between temperature tolerances and required response times would be determined in practice through experiments using the prototype. These experiments are likely to spark changes to the timing requirements as well as to the control algorithms — an important reason to build prototypes.

We believe that simulations of the software's environment are an essential part of rapid prototyping and that any language for prototyping real-time systems must support the construction of such simulations. Simulations are important because the actual environment of the intended system is often too dangerous or too expensive to risk while testing a prototype with unknown and possibly faulty properties.

Data types. The interface to Brain_Tumor_Treatment_System includes the abstract data type medical_his-

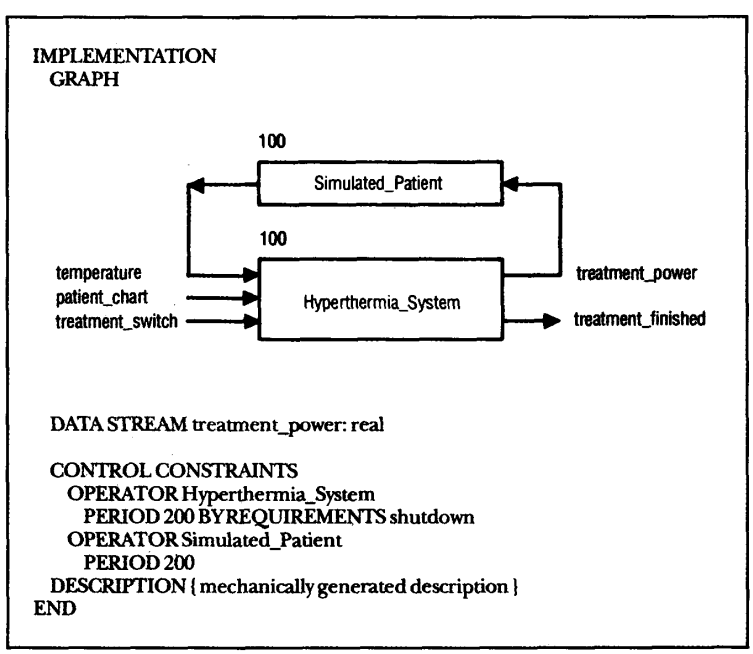


Figure 2. Lower level operators and the data streams connecting them for the microwave generator.

tory. Figure 3 shows the implementation part of medical_history. Only one operation of this type, Get_Tumor_Diameter, was needed for the prototype, although additional operations to create values of the type were later needed to exercise the prototype. We modeled the type as a tuple because a real medical history will contain many other components in addition to a tumor description. Because these attributes were not important for the prototype, they were not specified in detail.

We modeled the tumor description as a mapping from tumor location to tumor diameter because a patient can have more than one tumor and because the tumor size was the only attribute important for the prototype. We allocated most of the available time to the table lookup function map.Fetch because it involved searching, while extracting a fixed component of a record can be done in little time. In a mature system, the execution times for operations of built-in types such as tuple.Get would be obtained automatically from the software base.

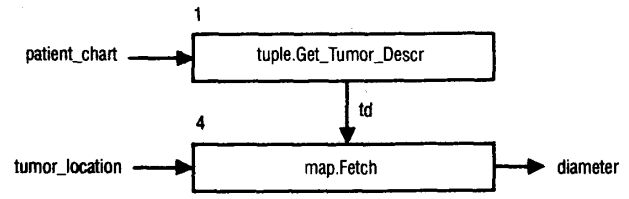
The data types tuple and map are built into PSDL, and their implementations are retrieved from the software base. A tuple is the Cartesian product of several component types, with symbolic names for the components. A map is a function from a finite subset of one data type to another data type, and is similar to a lookup table.

In the prototype, the Fetch and Get_Tumor_Description functions are primitive operations of the map and tuple types, so they need not be refined any further. Tuple is a parameterized family of types with a get_X operation for each component name X. We used the exception-control constraint to turn an exception of the built-in map type into a different exception meaningful for the medical_history type. This is an example of the kind of local adjustment commonly needed to adapt a reusable component to the needs of a particular prototype.

Lower level components. The decomposition step must be repeated for the components at lower levels until implementations can be retrieved from the software base or until the parts are implemented by functional simulations coded in the underlying language.

```
IMPLEMENTATION
  tuple[tumor_descr: map[from: string, to: real] ]

  OPERATOR Get_Tumor_Diameter
  IMPLEMENTATION
    GRAPH
```




```
    DATA STREAM td: tumor_description
    CONTROL CONSTRAINTS
      OPERATOR map.Fetch
      EXCEPTION no_tumor IF Not(map.Has(tumor_location, td))
  END
END
```

Figure 3. Implementation part of medical_history abstract data type.

In this section, we show the decomposition of one component at each level of the hierarchy, continuing until we hit the bottom. We have chosen a path down the hierarchy that terminates in a component implemented by a functional simulation coded in Ada. Figure 4 shows the PSDL description for Hyperthermia_System.

This example illustrates the use of an event-controlled timer, a conditional output, and conditionally activated operators. The treatment_time timer is reset (to zero) whenever the temperature drops below body temperature (such as at the end of a treatment session). The timer is restarted if the temperature is in the range for effective hyperthermia, and it is stopped if the temperature goes out of the range. The treatment_time timer records the treatment time and controls the transmission of the output treatment_finished from the Maintain operator.

The Maintain operator always produces the value True for the treatment_finished switch, while the Start_Up operator always produces the value False for the treatment_finished switch. Because the output of Maintain is conditional, the True value is transmitted only when the predicate giving the output condition is true. The initial False value persists until the conditional output is transmitted because treatment_finished is a sampled data stream (as are all the other data streams in this example). Both Start_Up and Maintain are triggered conditionally. The guards of these two operators are mutually exclusive, so only one is executed in any period.

The temperature going out of range is an unwanted event that is prohibited by the temperature-tolerance requirement. Constructing the prototype forces you to recognize that this event is possible and raises the question of how the system should behave if the event does happen due to some kind of malfunction. These events also let you compare control algorithms for the Maintain operator. In a typical development, you would prototype several control algorithms and compare their behaviors by monitoring the frequency of stop-timer events for treatment_time.

The specification for the Maintain operator at the next level of refinement is:

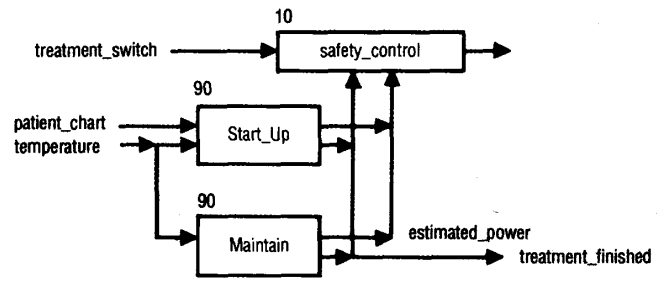
```
OPERATOR Maintain
SPECIFICATION
  INPUT temperature: real
  OUTPUT estimated_power: real,
    treatment_finished: boolean
  MAXIMUM EXECUTION TIME 90ms
    BY REQUIREMENTS
    temperature_tolerance
  DESCRIPTION
  { The power is controlled to keep the
    power between 42.4 and 42.6 degrees
    C. }
END
```

The Maintain operator is a specialized function not found in the software base. It is simple enough so that further decomposition is not useful, so we chose to implement it with a functional simulation in the

```
OPERATOR Hyperthermia_System

SPECIFICATION
  INPUT temperature: real, patient_chart: medical_history, treatment_switch:
    boolean
  OUTPUT treatment_power: real, treatment_finished: boolean
  MAXIMUM EXECUTION TIME 100 ms
    BY REQUIREMENTS temperature_tolerance
  MAXIMUM RESPONSE TIME 300 ms BY REQUIREMENTS shutdown
  KEYWORDS medical_equipment, temperature_control, hyperthermia,
    brain_tumors
  DESCRIPTION
  { After the doctor turns on the treatment switch, the hyperthermia system
    reads the patient's medical record and turns on the microwave generator
    to heat the tumor in the patient's brain. The system controls the power
    level to maintain the hyperthermia temperature (42.5 degrees C) for 45
    minutes to kill the tumorous cells. When the treatment is over, the system
    turns off the power and notifies the doctor. }
END

IMPLEMENTATION
  GRAPH
```




```
  DATA STREAM estimated_power: real
  TIMER treatment_time

  CONTROL CONSTRAINTS
    OPERATOR Start_Up
      TRIGGERED IF temperature < 42.4
        BY REQUIREMENTS maximum_temperature
      STOP TIMER treatment_time
      RESET TIMER treatment_time IF temperature <= 37.0

    OPERATOR Maintain
      TRIGGERED IF temperature >= 42.4
        BY REQUIREMENTS maximum_temperature
      START TIMER treatment_time
        BY REQUIREMENTS treatment_time, temperature_tolerance
      OUTPUT treatment_finished IF treatment_time >= 45 min
        BY REQUIREMENTS treatment_time

END
```

Figure 4. PSDL description for the Hyperthermia System operator.

underlying language. This decision was recorded in PSDL as follows:

```
IMPLEMENTATION Ada maintain
END
```

At the bottom level of the hierarchy, the PSDL implementation part gives the language the module is implemented in and the name of the implementation module.

An Ada implementation for the Maintain operator is:

```
PROCEDURE maintain (temperature:
  IN real; estimated_power: OUT real;
  treatment_finished: OUT boolean ) IS
  c: CONSTANT real := 10.0;
BEGIN
  IF temperature > 42.5 THEN
    estimated_power := 0.0;
  ELSE estimated_power := c *
    (42.5 temperature);
  END IF;
  treatment_finished := true;
END maintain;
```

This represents a conservative first design. It is very safe in the sense that it is very unlikely for the temperature to go too high. The algorithm is based on a very simple physical model assuming the tumor has no heat loss, so the rate of temperature increase is proportional to the applied power level. The model has the advantage of being very simple, so it can be implemented quickly. It is not very accurate, however, because the blood flow through the brain tissue will carry away excess heat. Because of this oversimplification, the first version of the control algorithm may not ever apply enough power to reach the hyperthermia temperature, violating the start-up time requirement.

This is typical of an iterated rapid prototyping effort. The initial version of the prototype will meet some but not all of the requirements. It is often fastest to construct a system by successive approximation — getting a quick skeleton in place that roughly approximates the required behavior — and then using that skeleton as a basis for planning further efforts.

Evaluating the prototype. You evaluate a PSDL prototype by running it through the PSDL preprocessor and then executing it on sample input data using the PSDL interpreter and debugger. First, you should test the prototype against the requirements and fix any design faults. You can also answer some of the questions that

prompted the prototyping effort. Once you believe the prototype behaves according to its specification, you demonstrate it to the users, who identify faults in the behavior. You record these faults and trace them back to the requirements. You can also measure the prototype's behavior to identify performance bottlenecks.

The PSDL execution-support system should perform static analysis and report certain design errors. The most important of these are violations of timing constraints, which the static scheduler discovers when it fails to find a valid schedule. Such errors can be caused by inconsistent constraints or insufficient resources. The first kind of violation is a design fault; the second is an error in estimating the computing resources needed. This second error can often be corrected by letting the scheduler use more processors.

Other errors detectable at this point include type inconsistency of interfaces and modules without implementations. The last kind of error should be reported but not be fatal. It should trigger the automatic creation of a stub module that roughly simulates the missing component through randomly generated output data of the appropriate types and a time delay conforming to the specified maximum execution time, if one is given. This should make it possible to simulate important attributes of a partially completed prototype, providing feedback on critical questions as early in the process as possible.

As in any other activity, human error is possible in prototype construction. You must execute the prototype with at least a minimal set of test data to make sure the prototype's behavior conforms to the design's intentions. The test cases produced in this stage should form the initial version of the demonstration to the user.

The prototyping effort started by identifying a set of questions. These questions can be answered in part by your choosing relevant input values and observing the prototype's behavior. Questions about the feasibility of performance constraints fall in this category. Sometimes the questions involve clarification of concepts that are not precisely defined in the requirements. In these cases, it is your responsibility to propose a precise version of the concept and to demonstrate to the user how the designer's proposed definitions affect the prototype's behavior.

It becomes the user's responsibility to examine the consequences of the definitions and of the requirements' current interpretation, as well as to judge whether the results are acceptable. It is also the user's responsibility to identify faults in the demonstrated prototype's behavior and to tell you what aspects of that behavior are not acceptable and why.

It is your (the designer's) responsibility to make sure that the full range of behavior that can be manifested by the prototype is included in the demonstration. It is also your responsibility to record the faults and to trace them to prototype components and to the requirements. The cross-reference facilities of the design database aid this stage of the process.

You also evaluate the example prototype to identify potential difficulties in the construction of the deliverable version of the intended system. This can include gathering statistics about the firing frequencies of the prototype's components to identify potential performance bottlenecks and evaluating the appropriateness of the design concepts used in the prototype, taking the feedback from the users into consideration. This evaluation may lead to new questions for the next iteration of prototyping and to the identification of critical subsystems that may be difficult to design or have tight performance requirements whose feasibility is questionable. Analyzing the prototype can also provide a cost estimate for the intended system.

In a serious prototyping effort for a hyperthermia system, you would determine at this point that the temperature requirements are difficult to meet and that a careful detailed analysis of the technical problems involved is needed. The next step in such a case would be to bring in expert consultants in the areas of biology, heat transfer, and control theory to develop an accurate simulation model of the thermal properties of the human brain and to propose and analytically investigate the stability and effectiveness of several control algorithms. You would prototype several alternative versions of the control algorithm in the Maintain operator and monitor the behavior of each version during execution to verify the experts' opinions.

Modifying the prototype. You must negotiate with the user a set of extensions and modifications to the requirements, based both on the faults identified in the demonstration and on cost and schedule estimates derived from the prototype. This process is coupled with reinterpretation and redefinition of some of the informal concepts in the original requirements. If such redefinitions are necessary, you should undertake another prototyping iteration and another demonstration and user review.

In the hyperthermia example, an additional requirement resulted from the possibility of a premature stop-timer event for treatment_time, indicating failure to meet the temperature_tolerance requirement. Executing the initial prototype detected this possibility because that prototype did not deliver a treatment_finished signal in less than one hour because it could not maintain the hyperthermia temperature at the required level. This situation was easily detected in a prototype demonstration, but it would be easy to overlook in other approaches to requirements analysis that do not involve computer aids.

Further consideration of this unwanted situation showed us that it could be due to either software or hardware malfunctions and that the absence of such a failure cannot be absolutely guaranteed. Consequently, we needed a new requirement to specify what "safe operation" means in the event of such a failure.

A strategy based on reusable software components is a promising, practical approach to rapid prototyping. Good modularity is especially important in prototyping because of the need to make many changes in a short time. A systematic method for prototyping is necessary but not sufficient for the rapid construction of prototypes for large real-time systems.

To make the process rapid, the method you use must be supported by a clear, simple, and expressive computational model supported by a matching language and automated prototyping environment. The same language must be used for prototype design and for software-base re-

trievals to gain the benefits of reusable software components. We have designed such a model and language, as well as the kernel of such an environment. The language has been applied to several examples and appears to be effective for designing and analyzing real-time systems. Construction of a prototype version of the environment is under way.

More basic research in two areas is also needed:

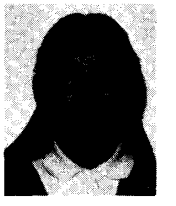
- Better methods for organizing and retrieving reusable components from the software base are important for the practical implementation of the prototyping method presented here. Previous work on retrieving components from a software base has been based on classifying components[6] rather than specifying their behavior. Some promising directions include software-base organizations based on adaptive generalization hierarchies and reusable-component retrieval based on specifications with a semantic canonical form.
- Computer-aided modification of prototype behavior is important for effective responses to user feedback during prototype demonstration sessions. Previous theoretical results on merging software versions[4] can be extended and applied to this problem. Efficient methods for implementing flexible interpreters with restarting checkpoints is another important area for further investigation. ❖

Acknowledgment

This research was supported in part by the National Science Foundation under grant CER-8710737.

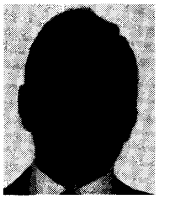
References

1. Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Trans. Software Eng.*, Oct. 1988.
2. Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Software*, March 1988, pp. 66-72.
3. R.T. Yeh, N. Roussopoulos, and B. Chu, "Management of Reusable Software," *Proc. Compcon*, CS Press, Los Alamitos, Calif., 1984, pp. 311-320.
4. V. Berzins, "On Merging Software Extensions," *Acta Informatica*, Nov. 1986, pp. 607-619.
5. Luqi and V. Berzins, "Execution Aspects of Prototypes in PSDL," Tech. Report 86-2, Computer Science Dept., Univ. of Minnesota, Minneapolis, Minn., 1986.
6. R. Prieto-Díaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, Jan. 1987, pp. 6-16.



Luqi is an assistant professor of computer science at the Naval Postgraduate School. She has also worked on software research and development at the University of Minnesota. Her research interests include software development tools, rapid prototyping, and languages.

Luqi received a BS in computational mathematics from Jilin University, China, and an MS and PhD in computer science from the University of Minnesota.



Valdis Berzins is an associate professor of computer science at the Naval Postgraduate School and at the University of Minnesota, where he is on leave. His research interests include software engineering and computer-aided design.

Berzins received a BS in physics, an MS in electrical engineering, and a PhD in computer science from the Massachusetts Institute of Technology.

Address questions to the authors at Computer Science Dept., Naval Postgraduate School, Monterey, CA 93943.